Linux 内核信号量分析

王 亚 军
（中国人民武装警察部队学院 计算机教研室，河北廊坊 065000）

摘 要：信号量机制是 Linux 内核对于访问同一个临界区的多个进程进行同步的有效手段。在深入分析内核信号量操作源代码的基础上，阐述了这种进程同步机制在内核中的实现，并基于嵌入式应用对其中存在的优先级倒转问题作进一步剖析。

关键词：信号量；临界区；优先级倒转
中图分类号：TP3 文献标识码：A 文章编号：（2008）03－0100－05

一、引论

一般来说，信号量是操作系统内核对访问共享资源的多个进程进行同步的一种手段。Linux 内核提供了两种类型的信号量：一种是由内核来支持，在系统空间实现的，供用户进程在用户空间使用的“用户空间信号量”；另一种是由内核提供的，供内核自身在系统空间使用的“内核信号量”，本文旨在分析这种信号量。

Linux 内核信号量又分为两种，一种是互斥信号量，这种信号量在一个时刻仅允许一个持有者，即使用者计数为 1，从而强制多个进程对临界区进行互斥访问；另一种是计数信号量，这种信号量的使用者计数可以大于 1，即同一时刻允许有两个或两个以上的进程访问临界区。如果一个进程试图获得一个已经被占用的信号量，该进程就可能会被推进一个等待队列进入“睡眠”状态，从而释放处理器，使处理器可以执行其他进程的代码。当持有信号量的进程释放信号量以后，在等待队列中睡眠的进程可能被唤醒，并试图获得该信号量以进入临界区。下面首先以 Linux－2.6.10 内核为例分析内核中运用信号量实现进程同步机制的源代码，然后结合嵌入式应用对信号量的使用中存在的优先级倒转问题作具体剖析。

二、Linux 内核信号量操作代码分析

在 Linux－2.6.10 内核中，内核信号量的数据结构类型是在 include/asm－i386/semaphore.h 中定义的：

```
struct semaphore {
atomic_t count;
int sleepers;
wait_queue_head_t wait;
};
```

其中的成员 wait 就是等待进入临界区的进程链表头。可以通过下面的方式静态地声明一个信号量（计数信号量或互斥信号量）：

```
static_DECLARE_SEMAPHORE_GENERIC(name, count);
```

其中 name 是信号量变量名，而 count 是使用者数量，当 count 值大于 1 时，该信号量就是计数信号量。创建互斥信号量还可以使用下面地快捷方式之一：

```
static DECLARE_MUTEX(name); /* name.count 初值为 1 */
static DECLARE_MUTEX_LOCKED(name); /* name.count 初值为 0 */
```

收稿日期：2008－07－15
作者简介：王亚军（1978－ ），男，黑龙江绥化市人，中国人民武装警察部队学院计算机教研室讲师，研究方向为嵌入式 Linux 系统设计。

?1994-2016 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

两者用途不同，但都是通过前面的宏定义__DECLARE_SEMAPHORE_GENERIC()来实现的。对信号量的 down()和 up()等操作函数也是在同一文件中定义的，函数头如下：

```
static inline void down(struct semaphore * sem)
static inline void up(struct semaphore * sem)
```

其中 down()操作的主要作用是先使 sem->count 减 1，减了以后的结果若为 0 或大于 0，表明当前进程可以进入临界区，则 down()操作结束；减了以后的结果若为负数，表明进程不能进入临界区，则进程在 down()操作中调用__down_failed()进入深度睡眠状态。函数__down_failed()以及有关的代码在 arch/i386/kernel/semaphore.c 中，该函数的主要目的在于调用另一个关键函数__down()，代码如下：

```
fastcall void sched_down(struct semaphore * sem)
{ struct task_struct *tsk = current;
DECLARE_WAITQUEUE(wait, tsk);
unsigned long flags;
tsk->state = TASK_UNINTERRUPTIBLE;
spin_lock_irqsave(&sem->wait.lock, flags);
add_wait_queue_exclusive_locked(&sem->wait, &wait);
sem->sleepers++;
for (;;){ int sleepers = sem->sleepers;
if (!atomic_add_negative(sleepers - 1, &sem->count)){
sem->sleepers = 0;
break;}
sem->sleepers = 1;/* us - see -1 above */
spin_unlock_irqrestore(&sem->wait.lock, flags);
schedule();
spin_lock_irqsave(&sem->wait.lock, flags);
tsk->state = TASK_UNINTERRUPTIBLE;
}
remove_wait_queue_locked(&sem->wait, &wait);
wake_up_locked(&sem->wait);
spin_unlock_irqrestore(&sem->wait.lock, flags);
tsk->state = TASK_RUNNING;
}
```

从函数__down()代码中可以看见进程进入休眠和唤醒后的工作过程：

(1)进程先调用 DECLARE_WAITQUEUE()创建一个等待队列的项 wait，该项结构中的成员指针 wait.task 指向这个进程的控制块结构 tsk->task_struct。

(2)进程将自己的运行状态修改为“睡眠”状态。

(3)进程调用 add_wait_queue_exclusive_locked()先将代表该进程的等待项 wait 中的成员 wait.flags 的标志位 WQ_FLAG_EXCLUSIVE 设置成 1，然后把自己加入到等待队列的尾部。

(4)在 for 循环中，进程再次判断自己是否可以进入临界区，如果可以，那么循环会退出，进程不会错误地进入休眠状态；如果不可以进入临界区，进程便释放自己占据的锁，并调用 schedule()进入睡眠状态，内核将调度其他进程投入运行。

(5)进程被唤醒以后，在 for 循环中，要判断自己所等待的条件是否真正的达成，还是存在虚假的唤醒。如果是虚假的唤醒，那么进程会再次进入睡眠状态。如果进程此次在 for 循环中成功地获取了信号量，那么循环会退出。

(6)进程将自己从等待队列中移出，并将自己的运行状态修改为可执行状态。

与 down()对应的函数 up()首先递增 sem->count，递增以后若结果为 0 或正数就调用函数__up_wakeup()唤醒在信号量等待对列中睡眠的进程。函数__up_wakeup()的功能是通过辗转调用文件 kernel\sched.c 中的函数__wake_up_common()实现的：

```
static void __wake_up_common(wait_queue_head_t *q, unsigned int mode, int nr_exclusive, int sync, void *key)
{ struct list_head *tmp, *next;
list_for_each_safe(tmp, next, &q->task_list){
wait_queue_t *curr;
unsigned flags;
curr = list_entry(tmp, wait_queue_t, task_list);
flags = curr->flags;
if (curr->func(curr, mode, sync, key) &&
(flags & WQ_FLAG_EXCLUSIVE) &&
!--nr_exclusive)  break;
}
}
```


?1994-2016 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

从函数_wake_up_common()的代码中可以看到调用该函数的进程唤醒等待队列中的睡眠进程的工作过程：调用该函数的进程用一个 for 循环扫描等待队列中从队首开始的睡眠进程。在每一遍循环中，调用 curr—> func()唤醒一个进程，如果唤醒成功，就判断代表该进程的等待项中的 curr 中的成员 curr. flags 的标志位 WQ_FLAG_EXCLUSIVE 是否为 1。如果该标志位为 0，那么继续唤醒等待队列中的下一个进程；如果该标志位为 1，那么是否继续唤醒别的睡眠进程将取决于其他条件，通常不再继续唤醒别的睡眠进程，比如互斥信号量作为内核同步的手段，不允许多个进程同时进入临界区，在这种情况下，等待队列中只有一个进程会被唤醒，其他进程要等到该进程退出临界区以后才可能被唤醒。默认情况下 curr—> func()是函数 try_to_wake_up()，该函数负责将欲唤醒的进程设置为 TASK_RUNNING 状态，并调用 activate_task()将此进程挂入可执行队列。如果被唤醒的进程优先级比当前正在执行的进程的优先级高，还要设置当前进程的 need_resched 标志，要求内核重新调度。

三、内核信号量中的优先级倒转问题剖析

从上述分析中可以看出，当一个进程试图获得一个已经被占用的信号量失败而被迫挂入等待队列时，内核仅是简单地将其加入到等待队列的尾部，而不管该进程是实时进程还是普通进程。当在这个等待队列上执行唤醒操作时，通常排在最前面的那个进程才能被唤醒，即这些睡眠进程是按照“先进先出”的原则被唤醒的，进程的优先级不起任何作用。这样，优先级低的进程就可能抢先优先级高的进程得到运行，造成了优先级倒转现象。

从上述代码分析中还可以看出，如果先申请信号量的普通进程进入了临界区，那么后申请信号量的实时进程就可能要挂入等待队列睡眠等待。由于信号量所保护的临界区内允许进程调度，已经进入临界区的普通进程在释放所占有的信号量之前还可能操作受阻而进入睡眠状态，这样进一步增加了实时进程的等待时间，加重了优先级倒转情况。

当把 linux 内核应用于嵌入式系统时，由于嵌入式系统通常具有实时性要求，所以应该基于嵌入式应用对内核信号量中存在的优先级倒转情况进行改进，以提高内核的实时性，这对于将 Linux 内核应用于具有实时性要求的嵌入式系统时是有意义的。

首先，需要把等待队列中采用的先进先出唤醒原则修改为高优先级实时进程优先唤醒原则。具体地说，对于那些因无法获得信号量而不得不进入等待队列睡眠的进程，如果是普通进程，就直接插入到等待队列的尾部（普通进程没有实时性要求，不必按照优先级排序），如果是实时进程，就按照其优先级插入到等待队列中，使得所有实时进程都排在普通进程的前面，高优先级的实时进程排在低优先级的实时进程的前面。这样，等待队列中的高优先级实时进程就能够先于其他进程被唤醒接受内核调度。

然后，可以运用“优先级继承”的思想解决占用信号量的进程与等待获得信号量的进程之间可能存在的优先级倒转问题，当有优先级高的进程在临界区外等待时，暂时把它的高优先级“借”给临界区内的进程，提高其竞争力，使其尽早得到调度，尽早退出临界区，从而释放所占用的信号量。linux 内核中使用的信号量绝大多数都是互斥信号量。当把 linux 内核应用于嵌入式系统时，由于嵌入式系统中一般不需要使用计数信号量，在这种情况下，就可以把计数信号量的操作机制从内核中裁除，只保留互斥信号量。而互斥信号量在一个时刻仅允许至多一个进程进入临界区，这样，内核信号量操作就大大简化，从而在不修改内核原有数据结构的前提下，就可以运用优先级继承的思想来解决优先级倒转问题。方法是对等待进入临界区的进程链表（即信号量的等待队列）进行修改，使得该链表中不仅包含等待进入临界区的进程，而且包含已经进入临界区的进程（至多一个），并将该进程固定置于队首。这样，当某个进程正占用临界区时，等待队列中的第一个元素就代表着该进程；而当临界区空闲时，等待队列中的第一个元素则为等待进入该临界区的优先级最高的进程。具体来说，如果一个进程 A 进入了临界区，那么 A 就成为等待队列中的队首元素，而另一个无法进入该临界区的实时进程 B 如果优先级高于等待队列中的其他进程，那么进程 B 在入队列时就会成为进程 A 的直接后继元素。此时，进程 B 还要判断自己的优先级是否高于进程 A，如果是，那么它就要把自己的高优先级“借”给进程 A，实现优先级继承。当进程 A 要退出临界区时，它首先把自己从等待队列中移出，然后判断自己的优先级是否继承自其他进程，如果是，就修改为原来的优先级，并安排一次进程调度，然后退出临界区。

?1994-2016 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

为了实现上述思想，需要对内核中的相关源代码进行修改。由于 linux－2.6 内核与 linux－2.4 内核关于信号量的操作机制是相同的，而嵌入式系统通常采用较低版本的内核，所以这里以 linux－2.4.0 内核为例阐述内核源代码的修改，需要修改文件 include/asm－i386/semaphore.h 中的函数 down()和 up()以及文件 arch/i386/kernel/semaphore.c 中的函数－down()和－up()。修改如下：

```
static inline void down(struct semaphore *sem)
{ asm__volatile_(
"jmp 2f \n"
"1: \n"
".section.text.lock, \"ax \" \n"
"2: \tcall down-failed \n \t"
"jmp 1b \n"
".previous"
:"=m"(sem->count)
:"c"(sem)
:"memory");
}
void down(struct semaphore * sem)
{ struct task-struct *tsk = current;
DECLARE-WAITQUEUE(wait, tsk);
tsk->state = TASK-UNINTERRUPTIBLE;
if(tsk->policy==SCHED-OTHER)
{add-wait-queue-exclusive(&sem->wait, &wait);
sem->sleepers++;
}
else{
spin-lock-irq(&semaphore-lock);
wait-queue-t *wqt;
struct list-head *tmp-list= ((sem->wait).task-list).prev;
struct list-head *tmp-list-stop= (sem->count ? &(sem->wait).task-list)
:(sem->wait).task-list).next);
while((*tmp-list != tmp-list-stop)&&(wqt=list-entry(tmp-list, wait-queue-t, task-list))&&(wqt->task->policy==SCHED-OTHER|| wqt->task->prio>tsk->prio))
tmp-list=tmp-list->prev;
wait->flags |= WQ-FLAG-EXCLUSIVE;
list-add-tail(tmp-list, &wait.task-list);
sem->sleepers++;
if(!sem->count)
{wqt= list-entry(tmp-list-stop, wait-queue-t, task-list);
if(wqt->task->policy==SCHED-OTHER|| wqt->task->rt-priority<tsk->
rt-priority)
{struct runqueue *rq;
unsigned long flags;
spin-lock-irq(&runqueue-lock);
del-from-runqueue(wqt->task);
if(wqt->task->policy==SCHED-OTHER) wqt->task->nice==1000;
else{ wqt->task->nice= wqt->task->rt-priority;
wqt->task->rt-priority= tsk->rt-priority;}
add-to-runqueue(wqt->task);
spin-unlock-irq(&runqueue-lock);
}
}
spin-unlock-irq(&semaphore-lock);
}
spin-lock-irq(&semaphore-lock);
for(;;)
{ if(sem->count&&((sem->wait).task-list).next==&wait.task-list)
break;
spin-unlock-irq(&semaphore-lock);
schedule();
spin-lock-irq(&semaphore-lock);
tsk->state = TASK-UNINTERRUPTIBLE;
}
atomic-dec(&sem->count);
spin-unlock-irq(&semaphore-lock);
tsk->state = TASK-RUNNING;
}
static inline void up(struct semaphore * sem)
{ asm__volatile_(
```


?1994-2016 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

```
"jmp2f \n"
"1: \n"
". section. text. lock, \"ax \" \n"
"2: \tcall up-wakeup \n \t"
"jmp 1b \n"
". previous"
:"=m" (sem-> count)
:"c" (sem)
:"memory");
}
void-up(struct semaphore *sem)
{ struct task-struct *tsk = current;
struct runqueue *rq;
unsigned long flags;
wait-queue-t *wqt= list-entry(((sem->wait)
. task-list). next, wait-queue-t, task-list);
spin-lock-irqsave(&semaphore-lock);
remove-wait-queue(&sem->wait, wqt);
sem-> sleepers--;
atomic-inc(&sem-> count);
if(sem-> sleepers)wake-up locked (&sem->
wait);
spin-unlock-irqrestore(&semaphore-lock);
if ( tsk-> nice< -20|| tsk-> nice> 19)
{ spin-lock-irq(&runqueue-lock);
del-from-runqueue(wqt-> task);
if (tsk-> policy== SCHED-OTHER)tsk->
nice+= 1000;
else {wqt-> task-> rt-priority= wqt-> task
-> nice;
wqt-> task-> nice= 0;}
add-to-runqueue(wqt-> task);
spin-unlock-irq(&runqueue-lock);
schedule();
}
}
```

四、结束语

综上所述，在 Linux 内核中，信号量及其操作是内核对访问共享资源的多个进程进行同步的关键，对系统的性能具有重要影响。当把 linux 内核应用于嵌入式系统时，由于嵌入式系统通常具有实时性要求，应该对信号量操作中存在的优先级倒转情况进行改进，以提高内核的实时性，这对于将 Linux 内核应用于具有实时性要求的嵌入式系统时是有意义的。

[参 考 文 献]

[1] 毛德操，胡希明. Linux 内核源代码情景分析[M]. 杭州：浙江大学出版社，2002.

[2] 陈莉君编著. Linux 操作系统内核分析[M]. 北京：人民邮电出版社，2000.

[3] 彭晓明，王强编著. Lniux 核心源代码分析[M]. 北京：人民邮电出版社，2002.

[4.] Robert Love. Linux Kernel Development[M]. China Machine Press, 2004.

[5] Michael Beck. Linux Kernel Internals[M]. Addison-Wesley, 1997.

[6] Daniel P. Bovet and Marco Cesati. Understanding the Linux Kernel[M.] O'Reilly & Associates, 2000.

[7] Gary J. Nutt. Kernel Projects for Linux[M]. Addison-Wesley, 2000.

[8] Scott Maxwell. Linux Core KernelCommentary. Guide to Insider's Knowledge on the Core Kernel of the Linux Code[M]. Coriolis, 1999.

[9] Remy Card, Eric Dumas, Franck Mevel. The Linux Kernel Book[M]. John Wiley &Sons, 1998.

[责任编辑 沈 勇]

Analysis of Linux Kernel Semaphore

Wang Yajun

(Computer Teaching-research Section The Chinese People's Armed Police Forces Academy, Langfang Hebei 065000)

Abstract: The mechanism of semaphore is an effective means of synchronizing many processes waiting to enter the same critical section. Based on analyzing the linux kernel codes, this paper expatiates on the realization of the mechanism of semaphore in linux kernel, and analyzes the priority inversion problem in this mechanism.

Key words: semaphore; critical section; priority inversion

?1994-2016 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net